下载APP

15 | LRU：在虚拟内存中页面是如何置换的？

2022-01-13 黄清昊

《算法实战高手课》 课程介绍 >

讲述：黄清昊
时长 18:23 大小 16.84M

你好，我是微扰君。

今天我们继续讲解操作系统中另一个常用的算法， LRU 算法（Least recently used），也就是最近最少使用页面置换算法。这是操作系统中常用的内存置换策略之一，在内存有限的情况下，需要有一种策略帮助我们把此刻要用到的外存中的数据置换到内存里。该算法也同样适用于许多类似的缓存淘汰场景，比如数据库缓存页置换、Redis 缓存置换等。

领资料

在开始讲解 LRU 算法本身之前，我们先来了解一下这个算法在操作系统中到底解决了什么问题。

操作系统的缓存淘汰

我们知道，计算机是建立在物理世界上的，底层的存储计算需要依赖许多复杂的硬件：比如内存、磁盘、纷繁的逻辑电路等。所以**操作系统的一大作用就是，通过虚拟和抽象为应用开发者提供了一套操作硬件的统一接口**，而分页机制的发明，就是为了不需要让用户过度操心物理内存的管理和容量。

通过虚拟内存和分页机制，用户可以在一个大而连续的逻辑地址和非连续的物理地址之间，建立起映射。其中，物理地址既可以真的指向物理内存，也可以指向硬盘或者其他可以被寻址的外部存储介质。

用户的程序可以使用比物理内存容量大得多的连续地址空间；而计算机在运行程序的时候，也不再需要把进程所有信息都加载到内存里，只加载几个当前需要的页就可以了。

极客时间

但是内存容量并不是无限的，**访问到不在内存中的其他页，硬件会触发“缺页”中断，操作系统会在内存中选出一个页，把它替换为需要访问的目标页**。这样我们才能访问到需要的数据。如果你对操作系统的内存管理机制感兴趣，推荐阅读CSAPP。

这种场景在各种需要缓存的系统中也很常见。比如知名的缓存中间件 Redis，就是利用内存读取数据的高效性，去缓存其他可能更慢的数据源的数据，以达到更快的 IO 速度，也用到了缓存置换算法。毕竟任何系统的存储空间都不是无限的，当我们缓存的数据越来越多，必然需要置换掉其中一部分数据。

而如何选择一个合适的页面（或缓存内容）来替换，就是我们今天的重点 LRU 算法主要讨论的内容。带着这个问题，我们开始今天的学习。

置换策略

具体怎么样的置换策略是更合理的呢？

我们主要观察的指标是**缓存命中率**：在整个系统的生命周期里对比数据访问时，可以直接从缓存中读到的次数和数据访问的总次数。

命中率越高，就代表越多数据可以直接从缓存中获取到，系统更少访问成本更高的存储，系统的整体时延就会降低。以操作系统为例，命中率高，就意味着我们发生缺页中断和从外存中获取数据的次数会减少，而访问内存的速度比访问外存要快得多，CPU 利用率当然也就会更高。

在操作系统中，页面置换策略其实有很多种，你可能也知道一些，比较常见的包括 FIFO（先进先出）、LFU（最不经常使用）、LRU（最近最少使用）等。页面置换算法，在上世纪六七十年代曾经是学术界讨论的热点。

其中 LRU 是实际应用最广的策略，因为它有着比较高的命中率并且实现非常简单，在虚拟内存系统中效果非常好。主要思想就是，**当我们需要置换内存的时候，首先去替换最久没有被访问过的数据，这能很好利用数据的时间局部性**，因为我们倾向认为最近被访问过的数据，在整个系统的生命周期里，有更大机会被访问到。

当然，LRU 也不都是最优的，比如在特定负载的网络代理缓存场景下，很可能使用 LRU 就并不是一个最佳选择，因为网络负载很可能在不同的时候变化很大。但是毫无疑问，LRU 在内存管理上有着绝佳的应用。

下面我们结合具体例子来看看这几个页面置换策略的区别。

时间局限性与页面置换算法

刚才提到的，时间局部性，是一个比较抽象的描述，为了更直观地讨论这些策略帮助你理解这个概念，这里用一个序列表示操作系统依次访问的页面，序列里的每个元素代表需要

访问的页码。假设整个物理内存最多只能放 3 页，当页数超过 3，并访问内存中不存在的数据，就会触发缺页中断。

我们把页面的访问序列叫做“引用序列”，之后的讨论都会建立在下面这个广为流传的引用序列例子上来展开，s 表示这个序列，s[i]表示第 i 次访问的页码：

```
1 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
```

复制代码

极客时间

随机页面置换算法

既然发生缺页中断时，我们需要确定一个主存中需要被替换的页，那么一种很自然而然的想法就是通过软硬件的随机发生器选择一个页面替换。

也就是第一种策略，随机页面置换算法。

这种思想非常简单也易于实现，但是很显然，这种算法并不令人很满意。因为它没有用到任何历史访问记录的信息，而历史信息是很有用的，也是我们唯一能用于优化命中率的依据。

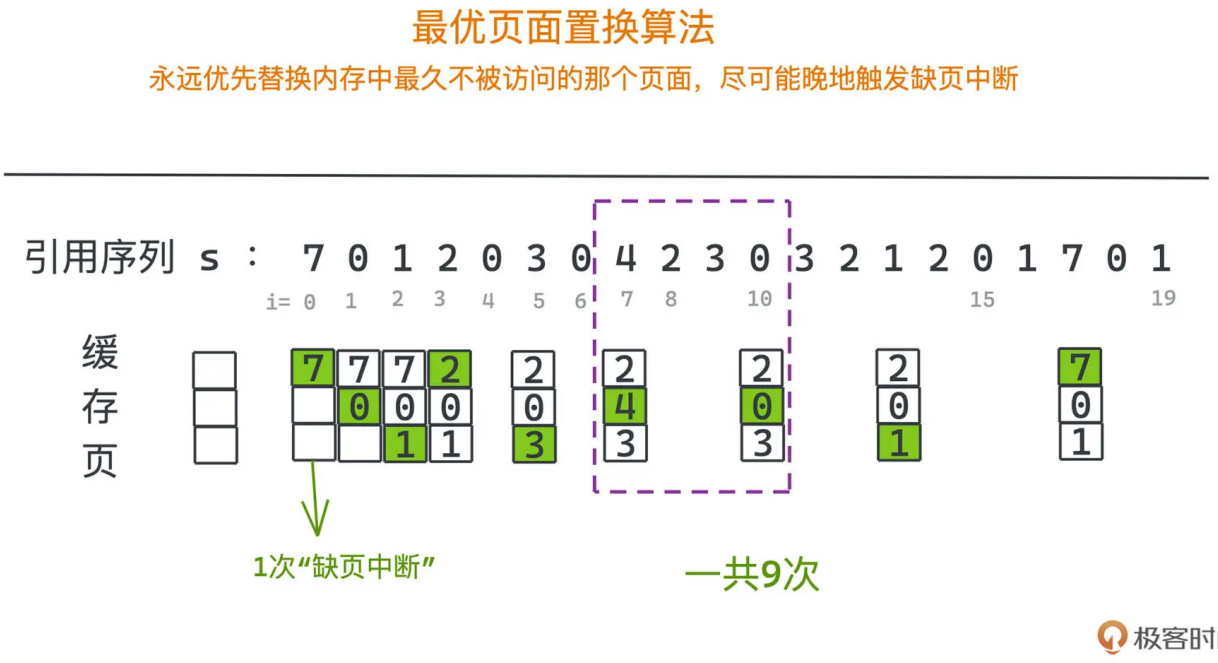
另外，这个算法导致同一个引用序列的产生的缺页中断次数是不稳定的，这会导致系统的性能不稳定，所以我们也不太会在实际系统中见到这样的策略。

最优页面置换算法

现在来看第二种策略，最优页面置换算法。这能让我们更好地理解为什么需要使用历史访问信息来帮助优化命中率。

从这个名字也能看出来，这个算法是一种最优的解法，但只是理论上存在的“上帝”算法，因为它的工作方式是，**在替换页面的时候，永远优先替换内存中最久不被访问的那个页面，尽可能晚地触发缺页中断。**

第一行就是引用序列，从高到低顺次排列的三个方块就代表三个缓存页，其中绿色的块代表新替换上的页面。



在例子中，我们访问 s[10] = 0 时，就可以把内存中的 4 替换掉，因为 4 在之后的访问中没有出现过。按照类似的策略，观察后续少出现的页码，尽量少触发缺页中断。所以一共只需要触发 9 次缺页中断。

很明显，**这并不是一个真的能被实现的算法，因为当运行程序时，并没有很好的办法去预测之后访问的页码是哪些**，唯一能做的就是尽量从历史的访问记录里推测出，哪些页码可能会很长一段时间不被访问。

总的来说，这个仅存在于理论上的算法主要的意义就是可以为我们衡量其他算法的好坏做一个参考。

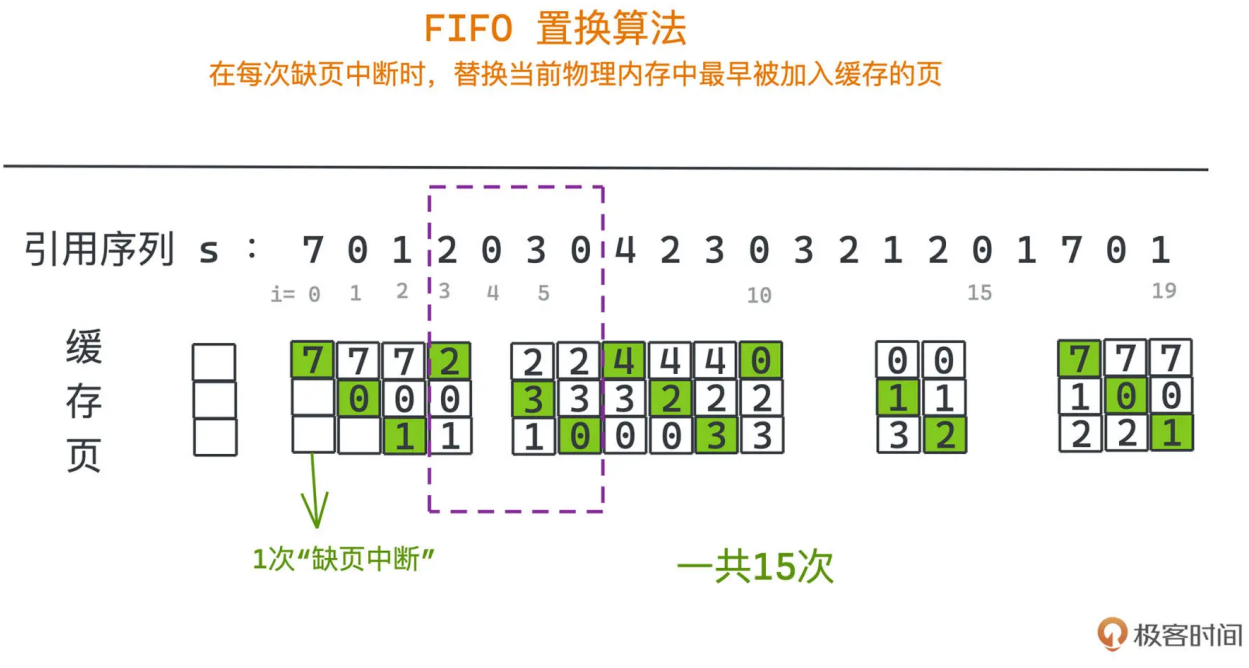
FIFO 算法

既然要利用历史记录，你是不是很自然想到放得久的数据先置换出去，也就是 First In, First out。这也就是我们要介绍的第三种策略，FIFO，先进先出算法。

先进先出，这个词相信你一点也不陌生，我们在第 3 讲介绍队列数据结构的时候也有提到，队列的基本特征就是先进先出。

在页面置换中，先进先出的策略是这样工作的：**在每次缺页中断时，替换当前物理内存中最早被加入缓存的页**。实现也很简单，可以通过一个循环链表来存储所有页。

这看似比较符合直觉，但在操作系统的实际应用中表现很差。我们结合刚才引用序列的例子来看，可以画出这样的示意图：



可以看到，我们进行了 15 次缺页中断，和最优解相比多了很多次置换。比如在序列是 s[4...6] = 0 3 0 的时候，这里的 0 因为出现的比较早，在 s[5]时被替换成 3 后，又遇到马上要读取 0 的情况，又要做一次缺页中断的操作。

所以对于符合直觉的 FIFO 算法，先加入的页面可能会被多次访问，如果经常让更早被加入但访问频繁的页面被淘汰，显然不是一个很好的策略。这意味着我们**不但要用历史数据，还要更好地设计利用的方式，让策略更接近于最优算法**。

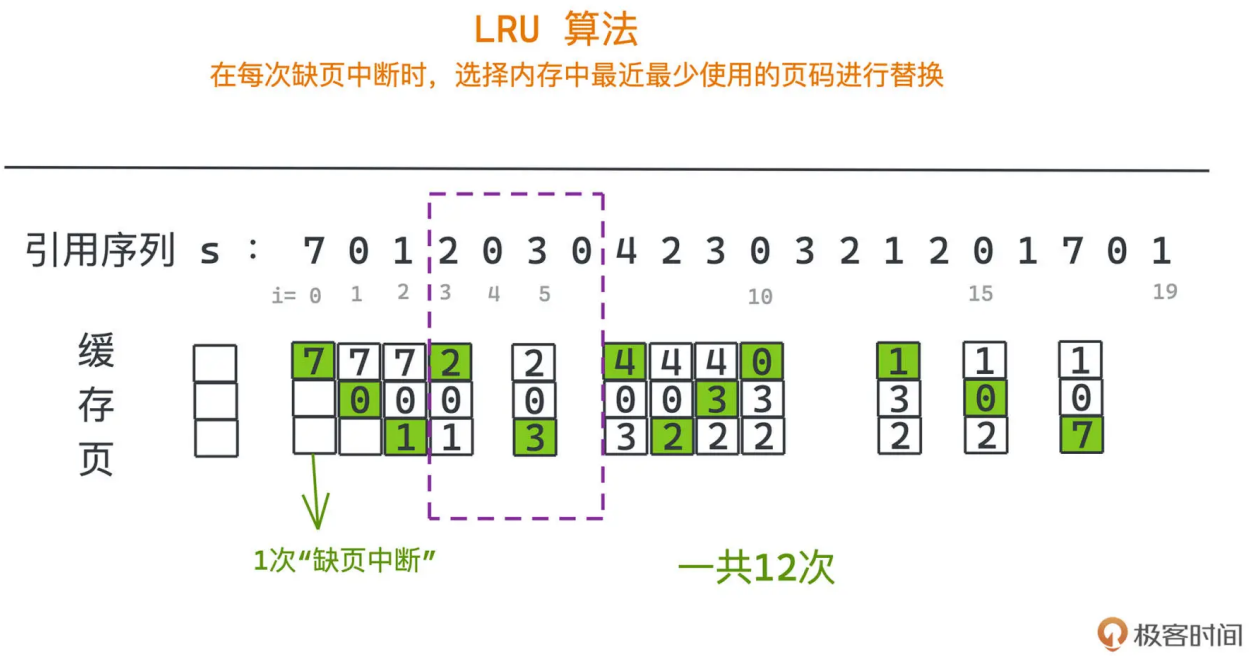
LRU 算法

今天的主角 LRU 最近最少使用算法，就是这样一种直观简单、实际检验效果也非常好的页面置换策略。

通过刚才的几个例子，结合你实际使用体验，会发现在操作系统的场景下，**引用序列有明显的局部相关性，每个出现的页码有比较高的概率会在相邻的一段时间里反复出现**。

上一个 FIFO 算法的一大失误就在于没有考虑局部性，当一个页码多次出现时，FIFO 并没有将这个信息记录并反映到淘汰页面的选择策略里，所以可能就会淘汰了一个近期出现过，但是之后又很快就会再次出现的页码。

既然我们不能预测未来，简单替换最早的页码也不好用，那么一种很自然的想法就是，**如果某个页码在过去访问过，就尽量晚点去淘汰它**。我们可以选择内存中最近最少使用的页码进行替换，这也正是 LRU 的策略。



比如在获取数据 3 时，我们在 LRU 中替换的是最久没有被访问的 1，而在 FIFO 中我们替换了 0。但是，0 刚访问过，理论上来说之后访问的概率也会更高，不过在 FIFO 策略下，因为 0 是最早进入的被替换了，就导致了后面访问 0 时产生了一次缺页中断。

相比于 FIFO，同样的例子我们只进行了 12 次缺页中断。采用 LRU 算法，大多数时候会比 FIFO 和随机策略有更好的性能。

实现思路

既然有了基本的想法，我们就要想办法高效地实现它了。

Linux 源码比较艰深且涉及的背景知识比较多，这里我们选择自己动手实现简单 LRU 的方式来进行源码级讲解。

对于从指定页获取数据的操作，可以用一个 HashMap 来模拟，可以用 key 代表页面号，用 value 代表页面中具体的数据。

所以问题可以更通用地抽象为设计一个数据结构，提供 get 和 put 两个接口。get 的时候输入一个 key，我们可以快速地访问 key 所对应的 value；put 的时候设置某个 key 对应的 value。**同时这个数据结构初始化时需要设定一个 capaticy**，当数据结构中的 key 数量

超过 capacity 时，按照淘汰最近最少使用元素的策略进行替换，使得数据结构中最多只有 capacity 个 key-value 对。

之所以说是一种更通用的抽象，就是因为这不止适用于页面置换场景，也适用于许多其他缓存场景，比如在 Redis 中你就可以看到类似的数据结构。

对应在页面置换场景下，每次缺页中断就相当于，对该数据结构进行了 key 为指定页码的 put 操作，而 capacity，自然就是物理内存能存放的最多页数啦。

为了高效地实现内存置换算法，我们大致有两个需求：

1. 找到一种数据结构，使得我们可以随时快速地找到最近最少访问的页码。
2. 在每次缺页中断替换页面的时候，维护这个数据结构不会带来太多额外的成本。

幸运的是，LRU 的 get 和 put 的操作都是可以在 O(1) 的时间复杂度内实现的。下面我们就来看看用什么数据结构可以满足这样的需求。

数据结构选择

先说 get 的部分，想在 O(1) 时间内根据 key 获取 value，很自然就会想到之前提到的哈希表。**通过维护哈希表，就可以在 O(1) 时间内判断某个 key 是否存在 LRU 中，或者访问到该 key 对应的 value**。

但我们还要保证最近最少使用的替换策略，要想办法记录下内存中数据访问的先后关系，才可以保证最近访问过的，要比更早之前访问过的后淘汰。一种很自然的想法就是维护一个基于最近访问时间的有序列表。

这当然有很多种实现方式。比如我们可以维护一个数组，从前到后依次存放最近访问过到最久没有访问过的 key。可是这样每次 get 的时候，我们就需要把数组中某个位置的 key 移动到数组的开始位置，并把后续的元素全部顺移一位。根据我们之前学到的知识，这样整体移动数组的操作的复杂度是 O(N)。

那么应该怎么做呢？

相信好好学习前面专栏的同学已经想到了，**双链表就可以实现，在 O(1) 内，删除节点并移动到指定位置的操作**。我们可以构建一个双链表，让链表元素按照访问时间顺序从前到后依次排列。

通过双链表 + 哈希表，就可以完美实现 LRU 基于最近访问时间排序的有序列表，这两种数据结构的组合非常常见，也有人称之为 LinkedHashMap。

代码实现

下面我们来看具体的代码，这次选择语法简明、性能优秀的 Golang 作为实现语言。

首先是基础的数据结构的定义。我们声明一个 LRU 的结构体，它包括以下三个成员标量：

1. size 是 LRU 的容量。
2. innerList 是一个 Golang 内置的双链表，来表示基于最近访问时间排序的序列。
3. innerMap 是一个 hashmap，Golang 同样提供了内置实现，我们主要用它来存储 key-value 对。

再定义一个 entry，表示在 innerList 中链表节点的数据结构，它同时记录了 key 和 value 的信息。

复制代码

```
type LRU struct {
  size      int
  innerList *list.List
  innerMap  map[int]*list.Element
}

type entry struct {
  key   int
  value int
}
```

然后实现 get，我们可以从 map 中基于 key 获取元素的信息，如果不存在，就直接返回不存在。

因为要保证 LRU 的链表始终按照最近访问时间排序，get 之后，我们当然需要把当前 key 对应的链表节点移动到链表的最开始，所以，在 hashmap 中，可以选择直接记录链表中的节点元素。借助于 Golang 内置的双链表，只需要调用 MoveToFront 就可以简短地实现这一逻辑。

复制代码

```
func (c *LRU) Get(key int) (int, bool) {
  if e, ok := c.innerMap[key]; ok {
    c.innerList.MoveToFront(e)
    return e.Value.(*entry).value, true
  }
  return -1, false
}
```

最后来看一下 put 操作。相比于 get 操作，代码逻辑会稍显复杂一些，同样代码会有两个支路。

复制代码

```
func (c *LRU) Put(key int, value int) (evicted bool) {
  if e, ok := c.innerMap[key]; ok {
    c.innerList.MoveToFront(e)
    e.Value.(*entry).value = value
    return false
  } else {
    e := &entry{key, value}
    ent := c.innerList.PushFront(e)
    c.innerMap[key] = ent

    if c.innerList.Len() > c.size {
      last := c.innerList.Back()
      c.innerList.Remove(last)
      delete(c.innerMap, last.Value.(*entry).key)
      return true
    }
    return false
  }
}
```

我顺着代码简单说明一下逻辑。如果 put 的元素在 LRU 中已经存在，首先根据 innerMap 找到链表中的节点，移动到最前，并修改其中的 value 值就可以了。同样，这种情况在页面置换的场景下并不会出现。

而如果 LRU 中不存在指定 key 对应的记录，我们就需要在链表开头插入该节点，并在容量不足的时候，淘汰一个最近最少使用的节点。这段逻辑其实也非常简单，由于链表已经是基于访问时间从近到远有序排列的了，我们直接删除链表尾部元素就行。

当然也需要同步在 innerMap 中删除对应的记录，否则就会有类似于内存泄漏的问题，innerMap 中的内存占用就会越来越多且永远没有机会释放。而我们需要做的也仅仅是在删除链表末尾节点前，记录下该节点对应的 key 的值，并调用 Golang 内置的 delete 方法。

到这里我们就实现了最近最少访问算法所需的数据结构，它广泛运用于在实际系统里，我自己在网络组件的开发中就用到了类似的数据结构，去主动关闭超过一定数量的闲置链接，节约了大量的系统资源。

总结

通过分页和虚拟内存的抽象，操作系统解放了用户对内存管理和容量的心智负担。当缓存的数据越来越多，如何选择一个合适的页面或缓存内容来替换，就是缓存置换算法的用武之地。

页面置换策略有多种，包括随机置换、FIFO、LRU 等，非常重要且常见的 LRU 通过利用引用列表的局部相关性，提高了页面的命中率。 LRU 的实现也并不是非常复杂，但需要对链表和哈希表有很好的理解才行，所以我们一定要认真打好数据结构和算法的基础。

LRU 不但是面试的常见考点，实际开发也相当常用。我在工作中就有手写过类似的数据结构，用于清理最久没有数据包上下行的非活跃链接。建议你用自己熟悉的语言实现一遍，在实现的时候，要记得多考虑一些并发场景下可能会产生的问题。

课后作业

留一个小小的问题给你：为什么在 LRU 的数据结构中，我们选择了双链表而不是单链表呢？欢迎你在留言区与我讨论。

分享给需要的人，Ta订阅本课程，你将得 20 元

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 14 | 调度算法：操作系统中的进程是如何调度的？

下一篇 16 | 日志型文件系统：写入文件的时候断电了会发生什么？

学习推荐

JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费

精选留言 (1)

写留言

Paul Shan
2022-01-13

LRU需要把中间的节点移动到头部，并且要删除尾部节点，单链表对这两个操作都需要O(n),双链表可以用O(1)完成。LRU组合了哈希表和双向链表各自的强项，因而提供了快速的查询和灵活的资源管理两项功能。

作者回复: 说的很对~